A Pattern-based Approach towards Expressive Specifications for Property Concepts

Geert Delanote, Jeroen Boydens and Eric Steegmans
KU Leuven, Dept. of Computer Science
Leuven, Belgium
{geert.delanote, jeroen.boydens, eric.steegmans}@cs.kuleuven.be

***Abstract*—In Object-Oriented programming, a significant effort has been made in recent years to increase the expressiveness of programming constructs for the production of code. Developers can implement more functionality in less lines, and with more compile-time guarantees. We have not seen such a similar evolution in the design and specification of code. Support for code specification remains a feature that is rarely integrated in the language itself (e.g., Eiffel), and is too often migrated to ad hoc language additions (e.g., annotations). The lack of such first-class, language-integrated support leads to (1) developers who are forced to write ad-hoc code specifications in a non-standardized manner, often ex-post and time-permitting, and (2) to situations in which other developers, who reuse that code, are tempted to read the code itself (if available) rather than the specification, in order to understand what the code actually does. In this paper, we take an evolutionary approach to language-integrated specification constructs, with the ambition to enhance the overall expressiveness of specifications in object-oriented languages. We start from existing best practices and propose improvements through specification patterns that not only enhance the expressiveness of specifications, but also aid developers in specifying their code through concrete "structures" in order to avoid ad-hoc, non-standardized specifications. Finally, we also propose language constructs that help aim to increase the level of abstraction, by shielding developers from boilerplate specification as much as possible.**

Keywords*—*Pattern; Specification; Property; Language Construct.

I. Introduction

Object oriented programming languages use classes as abstract data types [1][9]. A class is a blueprint for a collection of objects with identical characteristics and behavior. Encapsulation hides the technical details of the data fields used in the implementation to describe those characteristics. Generally, several requirements have to be enforced for those characteristics. Examples of such requirements are: the balance of a bank account must not exceed the credit limit, a single transaction must not change the balance with more than €1000 and the holder of a bank account must be adult. Programming language constructs lack expressiveness to describe those requirements in an integrated way.

In this paper, we present a pattern to implement characteristics with their requirements in Java. We identify different kinds of requirements and show how they are implemented by the pattern. The pattern is only worked out for properties in this paper. However, with some adaptations to meet the specific needs, the pattern can also be used for (bidirectional) associations. We will show how the pattern improves the quality of the code. Finally, we will also show a new language construct that can replace the pattern.

This paper is structured as follows. Section II defines the quality objectives we want to improve. Section III presents some general programming principles to improve the quality. The different kind of requirements related to the development of properties are described in Section IV. Section V shows how the different requirements are developed in the pattern and how the pattern improves the quality of the code. In the last paragraph, a Language Construct that improves the expressiveness of a programming language is presented. Section VI presents related work. We conclude in Section VII with a view on future research roadmap.

II. Objectives

Object-Oriented languages were initially built to increase the quality of software applications [6]. Software quality is a combination of several factors [1]. Using software patterns is an important way to increase the quality of software systems [2]. We believe that more expressive language concepts can help to further improve the quality of software systems. Therefore, we believe that, as a second step, patterns should be transformed as much as possible into language concepts to avoid known drawbacks from patterns like implementation overhead (boiler plate code) and reusability (the programmer is forced to implement the pattern over and over again) [5]. Software quality factors break down in external and internal factors. In this paper, we mainly focus on the internal factors: factors perceptible for programmers. In the end, only external factors count, but the internal factors make it possible to obtain them [1]. We have centered the specification and development of our pattern along the following quality factors.

O1 - Correctness. Software must perform its task as defined by the specification. The pattern defines specific methods to work out the different aspects of the implementation of a characteristic forcing the developer to think about each aspect in isolation.

O2 - Extendibility. Software must be adaptable to future changes of the specification. These changes can be in space (through adding a subclass that redefines some aspects) or in time (changes to specification in the future). The pattern provides the necessary hook methods to be able to change the specification easily. The pattern also guides the developer to specify and implement each aspect only once.

O3 - Testability. Testing the correctness of software must be as easy as possible. Different aspects of the implementation

of a characteristic are worked out in separate methods. The methods are designed in such a way that they can be tested in isolation.

O4 - Understandability. A programmer must understand as easy as possible the source code of a software system. Dividing a big problem into smaller problems is a well-known strategy to make a problem easier to understand. The pattern separates the code, the developer has to write, from boilerplate code to make the code more readable.

O5 - Reusability. Software should be usable in different applications. Extendibility already mentions the provided hook methods to change the specification easily. These methods make it also easy to reuse the software in a (slightly) adapted form in another application.

O6 - Expressiveness. The ease for a developer to write software. By forcing the developer to implement the different methods, the pattern also guides the developer through the different aspects of implementing the characteristic. This way the developer can think more on *what* must be implement instead of on *how* he can accomplish it.
We raise the ambition level for each of these objectives when compared to the current state of the practice (throughout this paper, we will refer to these objectives using their codes). Our language concept, resulting from this pattern, also meets these objectives.

III. Principles and Notation

In this paper, we will follow the principles and notations introduced in the book *Object Oriented Programming with Java* [9]. The book presents three different paradigms to deal with exceptional circumstances: nominal, defensive and total programming. Nominal programming uses preconditions to prohibit method invocations under exceptional conditions. Defensive programming uses exceptions to signal that methods have been invoked under exceptional conditions. Total programming turns exceptional conditions into normal conditions. We have chosen to work out the examples in this paper in a defensive way. Transformation to the other paradigms is straightforward.

P1 - Inspector-mutator principle. An important principle is that we make a clear distinction between *inspectors* and *mutators*. Inspectors return information about the state of some objects. Mutators change the state of some objects. We try to avoid methods that combine both aspects: inspectors should not change the observable state of one or more objects and mutators should not return a result. We further distinguish between basic queries and derived queries. A basic query returns part of the state of an object. The state of an object is determined by the set of all basic queries. The result of derived queries and the effect of mutators is directly or indirectly specified in terms of basic queries. This principle improves the quality factors described in objectives O2, O3, O4, O6.

P2 - Steady versus Raw state. We distinguish between a *steady state* and a *raw state* for objects. An object in steady state satisfies all its invariants. An object in raw state is not guaranteed to satisfy all its invariants. Unless explicitly stated otherwise all objects must be in the steady state upon entry to and exit from a method. The general principle in defining methods is to assume that all objects are in the steady state. However, in some specific situations we want to use methods that involve objects that are in raw state. A typical example of such a situation is construction. While not yet in a steady state we sometimes want to use other methods during the initializing process. This principle acts as a contract between the developer and user of a method and by doing so helps to improve quality factors O1, O3, O4, O6.

P3 - Liskov Substition Principle. Changes to the definition of inherited methods must obey the Liskov Substition Principle [3]. Broadly speaking, the principle states that it must always be possible to substitute an object of a superclass by an object of its subclasses. Next to changes to the signature of inherited method, changes to the specification can be made if the superclass does not provide a deterministic specification of the result. Non-determinism plays a crucial role in our pattern. This principle supports all objectives O1-O6.

P4 - Complete business logica. All business rules should be worked out in specification and implementation. For enforcing business rules we never rely on the underlying persistence level. Integrity constraints, non-null constraints, foreign keys, etc., can be enforced by a database, but should (also) always be enforced by the application. This principle improves objectives O1, O3, O4, O5.

Notation. In Java, the contract of a class is worked out in documentation comments, which can be processed by javadoc [15]. Tags structure the different pieces of the specification in the documentation. The specification of a class is described both formally and informally. The informal specification is written in natural language, while Java boolean expressions are used to write the formal specification. Writing the specification formally improves the objectives O1, O3, O4. The following tags are used in the code snippets throughout this paper:

- @basic: denotes a basic query
- @effect: specifies the semantics of a mutator in terms of another mutator
- @invar: denotes a class invariant
- @post: specifies a postcondition of a mutator
- @raw: denotes an object in a possible raw state
- @return: specifies the result of a derived query
- @throws: specifies the exception that must be thrown when the specified assertion evaluates to true

IV. Requirements

Business rules can be generally described using three types of requirements: (1) Value Requirements, (2) State Requirements and (3) Transition Requirements.

Value Requirements. (VR) These requirements are used to specify the most basic kind of business rules in that they restrict the range of values that a characteristic, property or association, can have. Meeting its value requirements is a necessary condition for an object to be in a steady state (P2). A value requirement never takes into account other characteristics of the class at stake. For *properties* a value requirement restricts the set of values offered by its type further. A value requirement is for instance used to enforce that the credit limit of a bank account always needs to be below 0. For *associations* a value requirement restricts the multiplicity of

an association. The requirement that a bank card always needs to be linked with a bank account is a value requirement (this type of requirement is also know as *existential dependency*). Considering generalization/specialization, a redefinition that restricts the kind of objects a specialization can be linked with is a also a value requirement. The requirement that current accounts and savings-accounts, specializations of bank account, have the right specialization of bank cards attached to it is enforced with a value requirement.

State Requirements. (SR) Mostly, business rules restrict possible values for a characteristic when considered in combination with values from other characteristics. State requirements are by nature *symmetric*. A state requirement involving characteristics α and β is always a state requirement for both characteristics. Meeting its state requirements is the other necessary condition for an object to be in a steady state (P2). The union of all value requirements and state requirements describe all the invariants of a class. The business rule stating that the balance of a bank account must never be below the credit limit is specified by a state requirement.

Transition Requirements. (TR) These very specific requirements specify the business rules that restrict the evolution of values of characteristics. It's perfectly possible that a (new) value for a characteristic meets all value and state requirements but is not acceptable because of the current state of the object. The business rule imposed by a bank limiting the amount of money that can be withdrawn from a bank account is transition requirement. Although 1.000 euro is a correct balance, it's not an acceptable balance after a withdraw operation when the current balance is 10.000 euro and the withdraw limit is 5.000 euro.

In the remainder of this paper, we will show how our pattern implements the value, state and transition requirements. We will prove how distinguishing between these kinds of requirements together with the pattern with its specific methods meets the targeted objectives. We will also discuss how Java (and other object-oriented programming languages) can be extended with new language concepts to capture value, state and transition requirements.

V. Properties

In this section, we build the pattern for properties, step by step. These steps already give a good indication of what an iteration of the development process can consist of. It is possible to elaborate the different requirements independent of each other (O1, O3, O4, O6). Typically a pattern contains boilerplate code, we will highlight those parts in the code listings. The code editor should generate this code (O1, O6). In Eclipse [17], custom templates can be defined to generate skeletons of methods. Due to space limitations we omit the informal specifications. Steegmans illustrates in [9] how informal specifications should be added.

The example used throughout the next paragraphs describes a class of bank accounts. Each bank account has two characteristics, namely a balance and a credit limit. Both characteristics are decimal values and the balance must never be less than the credit limit. The amount of money that can be deposited or withdrawn in a single transaction must be restricted to 1000. To explain the pattern in the context of inheritance, we introduce a class of junior bank accounts, a subclass of bank accounts. The balance and credit limit of junior bank accounts are restricted to integer values. At the level of the subclass, two new characteristics are introduced: each junior bank account has an integer value as upper limit and a blocked state (boolean). While the credit limit can no longer be less than -1.000, the upper limit must at least be 1.000 and must not exceed 10.000. The upper limit is an immutable characteristic. Of course, the balance is not allowed to exceed the upper limit.

Representation. Each observable characteristic is part of the state of an object and is revealed by a basic query. The basic query can be compared with the getter from Enterprise JavaBeans (EJB) [10], [16]. The return type of the basic query reveals the chosen type for the characteristic. The characteristic can internally be stored using one or more *instance variables* with the same or different types. The implementation of the basic query has to perform necessary transformations between stored and observable information. Like EJB, we introduce also a *setter* to change the characteristic to a given value. The basic query and this setter are the only two methods that are allowed to access the instance variables that represent the characteristic. By consequence, we limit the optional transformations between internal representation and observed value of a characteristic to these methods (O1 - O6). When clients of a class are not allowed to change the value of a characteristic directly and need to manipulate the characteristic through more complex mutators, the latter mutators must be implemented in terms of this setter. When there exists a default value for the characteristic then that value is always explicit added to the declaration, even if that value is the default value of the type of the internal representation. Thus, absence of a default value in the declaration means this characteristic must always be initialized during construction (O4). Figure 1 illustrates the internal representation with default value, basic query and setter for the characteristic balance. As the stored and observed values are equal the implementation of both methods is trivial. The basic query is annotated `@Raw` because we also want to be able to observe the state of the property balance when the object is not in a steady state.

```
1  private BigDecimal balance=BigDecimal.ZERO;
2
3  /**
4   * Return the balance of this bank account
5   */
6  @Basic @Raw
7  public BigDecimal getBalance(){
8    return balance;
9  }
10
11 /**
12  * Set the given balance as the balance of
13  * this bank account
14  * @post  new.getBalance() == balance
15  */
16 public void setBalance(BigDecimal balance){
17   this.balance = balance;
18 }
```

Fig. 1: Representation of the property balance

Value Requirements. For each property, *a Boolean inspector* is introduced to validate the value requirements. This inspector is the only place where these requirements are specified and implemented (O1 - O6). Because the result of this inspector is by definition independent of the state

```java
/**
 * @return if ((creditLimit==null) ||
 *                  (creditLimit.signum() >= 0))
 *          then result == false
 */
public static boolean isProperValueForCreditLimit(
                BigDecimal creditLimit){
  return  (creditLimit != null) &&
          (creditLimit.signum() < 0);
}

/**
 * @post  new.getCreditLimit() == creditLimit
 * @throws IllegalArgumentException
 *          !isProperValueForCreditLimit(creditLimit)
 */
public void setCreditLimit(BigDecimal creditLimit)
                throws IllegalArgumentException{
  if (! isProperValueForCreditLimit(creditLimit))
    throw new IllegalArgumentException();
  this.creditLimit = creditLimit;
}
```

Fig. 2: Value Requirement of the property credit limit

of the object, the inspector is a class method (`static` in Java). By convention, the name of the inspector checking the VR for a property α is `isProperValueFor`α`(T` α`)` (O4, O6). According to P4, all business rules must be enforced in the application. Calling the setter with an actual argument that violates the VR is an exceptional situation and must be signaled. The setter is adapted accordingly. Figure 2 illustrates the inspector and setter for the characteristic credit limit. The specification of the inspector is worked out in a non-deterministic way. It specifies only which values are certainly not acceptable as value for the credit limit of a bank account. Notice however that the signature of the inspector `isProperValueForCreditLimit()` implies that only true or false can be returned as result. This way subclasses can decide to further restrict possible values or to explicitly confirm what values are always acceptable (O2, O5).

State Requirements. A state requirement describes a constraint that restricts acceptable value combinations of characteristics. Each SR is described by *a Boolean inspector*. This inspector is again the only place to specify and implement the SR at stake (O1 - O6). The inspector has an argument for each characteristic involved in the SR. Thus, this inspector is also a class method. Obviously, the value from each involved characteristic must meet the VR to have an acceptable combination of values. By convention, the name of a SR involving properties α and β is `isProper`$\alpha\beta$`(T1` α`, T2` β`)` (O4, O6). Each characteristic can be involved in multiple SR. We will illustrate in the paragraph about transition requirements how these inspectors are integrated in the setter. Figure 3 illustrates the SR between the properties balance and credit limit. The specification of this inspector is also non-deterministic; it is, however, also possible to close the specification and make it deterministic.

Invariant. The invariants for a class are described by the union of all VRs and SRs. We say that a characteristic α meets its invariants if it meets the VR and all the SRs it is involved in. For each characteristic α, we introduce *a Boolean inspector* to check whether a given value meets its invariants

```java
/**
 * @return  if (!isProperValueForBalance(balance))
 *          then result == false
 * @return  if (!isProperValueForCreditLimit(
 *                              creditLimit))
 *          then result == false
 * @return  if (creditLimit.compareTo(balance)>0)
 *          then result == false
 */
public static boolean isProperBalanceCreditLimit(
    BigDecimal balance, BigDecimal creditLimit){
  return isProperValueForBalance(balance) &&
      isProperValueForCreditLimit(creditLimit) &&
      (creditLimit.compareTo(balance) <= 0);
}
```

Fig. 3: State Requirement between balance and credit limit

with respect to the current state of the object. By convention, the name of this inspector is `canHaveAs`α`(T` α`)`. As this method is the sum of the VR for α and all SRs where α is involved in, this method can be generated as a whole (O1, O4, O6). With respect to the property α, the object is in a steady state if the current registered value for α meets its invariants. The inspector `hasProper`α`()` specifies the invariant for α. This method can also be generated (O1, O4, O6). Figure

```java
/**
 * @invar hasProperBalance()
 */
public class BankAccount {
...
/**
 * @return result==canHaveAsBalance(getBalance())
 */
@Raw
public final boolean hasProperBalance(){
  return canHaveAsBalance(getBalance());
}

/**
 * @return if (!isProperValueForBalance(balance))
 *          then result == false
 * @return if (!isProperBalanceCreditLimit(
 *                  balance, getCreditLimit()))
 *          then result == false
 */
@Raw
public boolean canHaveAsBalance(BigDecimal balance){
  return  isProperValueForBalance(balance) &&
          isProperBalanceCreditLimit(
              balance, getCreditLimit());
}
}
```

Fig. 4: Invariant from the property balance

4 illustrates these methods for the property balance. The inspector `canHaveAsBalance` is non-deterministic to allow new SRs in future subclasses (O2, O5). If new SRs are undesired the developer of this class can declare the inspector

`final` and make the specification deterministic. The inspector specifying the SR between balance and credit limit will be used in both the invariant inspector for balance and credit limit. By writing each SR in its own inspector, we avoid the need to duplicate that specification and implementation (O1 - O6). Both inspectors are annotated `@Raw`. Indeed, even when an object does not meet its invariants we want to be able to check if a given value meets its invariants.

Transition Requirements. A new value for a property must at least always meet the requirements described by the invariant. But often specific requirements restrict possible transitions when we take into account the current value of that property. The *Boolean inspector* `canHaveAsNew`α`(T` α`)` checks whether the given α is an acceptable new value with respect to the current state of the object (O2, O3, O4, O5, O6). First of all, the new value must meet its invariants. The extra TRs are added on top of them. The setter uses this inspector as guard for new values. Figure 5 illustrates this inspector and

```
1  /**
2   * @return  if (!canHaveAsBalance(balance)
3   *          then result == false
4   * @return  let BigDecimal difference =
5   *        getBalance().subtract(balance).abs() in
6   *          result == difference.
7   *                    compareTo(MAX_DELTA)<=0
8   */
9  public boolean canHaveAsNewBalance(
10                      BigDecimal balance){
11   return  canHaveAsBalance(balance) &&
12           (getBalance().subtract(balance).abs().
13                    compareTo(MAX_DELTA)<=0);
14 }
15
16 /**
17  * @post new.getBalance() == balance
18  * @throws IllegalArgumentException
19  *          !canHaveAsNewBalance(balance)
20  */
21 public void setBalance(BigDecimal balance)
22             throws IllegalArgumentException{
23   if (!canHaveAsNewBalance(balance))
24     throw new IllegalArgumentException();
25   this.balance = balance;
26 }
```

Fig. 5: Transition requirement of the property balance

the adapted setter. Often a public setter will not be desired, mutators like `withdraw` and `deposit` are preferred above `setBalance`. It suffices to change the access modifier to `protected` (`private` doesn't allow subclasses to define custom mutators) and custom mutators can easily be specified in terms of this setter.

Construction. Construction is an event with very specific semantics. After the complete construction process an object must be in a steady state. Because that is also the first state of the object we don't have the compare the initial value of a characteristic with its previous value (there isn't one). Even when there is value assigned in the declaration to the instance variable, we don't consider that value as a 'previous' value. An immediate consequence is that we can't use the setter in the constructor. Because we still want to restrict the manipulating of the instance variable(s) to a single method we need to introduce a more basic setter: `register`α`(T` α`)` (O1, O2). Figure 6 illustrates the basic setter for the property balance. Because this setter will be used in the constructor only the VR

```
1  /**
2   * @post  new.getBalance() == balance
3   * @throws IllegalArgumentException
4   *     !isProperValueForBalance(balance)
5   */
6  @Raw
7  protected void registerBalance(
8                        BigDecimal balance)
9              throws IllegalArgumentException{
10   if (!isProperValueForBalance(balance))
11     throw new IllegalArgumentException();
12   this.balance = balance;
13 }
```

Fig. 6: Basic setter for the property balance

is checked in this setter. This setter is also necessary when we want to introduce a complex mutator that manipulates two via SRs related properties. The developer will have to build a custom transition checker for that mutator but that is a rather trivial task as all building blocks are available. Indeed, each VR and SR is specified in its own inspector (O1, O2, O5, O6).

A steady state after construction means that all VRs and SRs must be met. Unfortunately, we can not use the inspector `canHaveAs`α`(T` α`)` because this inspector assumes all other properties β, γ,... already have their value. As there is no order in the different assertions of the specification, using them is impossible. So, we are forced to repeat the invariant

```
1  /**
2   * @effect  registerBalance(balance)
3   * @effect  registerCreditLimit(limit)
4   * @throws  IllegalArgumentException
5   *     !isProperBalanceCreditLimit(balance,
6   *                                 creditLimit)
7   */
8  public BankAccount(
9        BigDecimal balance,BigDecimal creditLimit)
10                 throws IllegalArgumentException{
11   if (!isProperBalanceCreditLimit(balance,
12                                   creditLimit))
13     throw new IllegalArgumentException();
14   registerBalance(balance);
15   registerCreditLimit(creditLimit);
16 }
```

Fig. 7: Construction of a bank account

conditions in the specification of the constructor. Fortunately, we can describe the semantics of the constructor in terms of other mutators, more in particular the basic setter, through the `@effect`-tag. This way we reduce the complexity of the specification and implementation (O1, O4, O6). So we only

need to list all SRs in the `@throws`-clause. Figure 7 illustrates the constructor for the class of bank accounts.

Inheritance. On the one hand, a subclass can specialize a superclass. The subclass can adjust the semantics of inherited features. The Liskov Substitution Principle (LSP) [3] acts as a guideline to describe allowed adjustments. On the other hand a subclass can extend the superclass with new features. We will illustrate how the pattern copes with specialization and extension. A subclass may want to redefine the VR of a property. This means we need to be able to override the inspector checking the VR. Because the inspectors checking the VR are class methods and Java does not allow to override `static` methods the way a VR is implemented in the pattern needs to be adapted. Clearly, these inspectors need to be instance methods but on the other hand they have class semantics as their result is defined independent of the state of the object. Therefore, we move these methods to a static inner class. This static inner class implements the Singleton Pattern [2]: the object of the static inner class represents the outer class. The marker interface [4] `ClassObject` designates the static inner class. Figure 8 illustrates the inner class for the class of bank

```
1  public class BankAccount {
2    public static class COBankAccount
3                        implements ClassObject{
4    private static COBankAccount instance;
5
6    protected COBankAccount(){}
7
8    public static COBankAccount getInstance(){
9      if (instance == null)
10       instance = new COBankAccount();
11     return instance;
12   }
13
14   public boolean isProperValueForBalance(...)
15   {...}
16
17   public boolean isProperValueForCreditLimit(...)
18   {...}
19
20   public boolean isProperBalanceCreditLimit(...)
21   {...}
22   }
23 }
```

Fig. 8: ClassObject inner class for the class BankAccount

accounts. The methods with class semantics can be moved without modification to the inner class. The specification and implementation of the instance inspectors using these methods can easily access them through the singleton object. A first advantage of moving the inspectors with class semantics into an inner class is that although they are instance methods can easily be identified as methods with class semantics (O4). A second advantage is that they make it impossible for the developer to use the state of the object erroneously (O6). A third advantage is that it is still possible to test these methods without needing an instance of the outer class (O3). If class B is a subclass of A, then the inner class of B must be a subclass of the inner class of A to be able to override methods from the inner class of A. Figure 9 illustrates the redefinition of the

```
1  public class JuniorBankAccount
2                        extends BankAccount{
3    public static class COJuniorBankAccount
4                        extends COBankAccount{
5    /**
6     * @return  if (!super.isProperValueForBalance(
7                                    balance))
8     *          then result == false
9     * @return  if (balance.scale()!=0)
10    *          then result == false
11    */
12   @Override
13   public boolean isProperValueForBalance(
14                       BigDecimal balance){
15     if (!super.isProperValueForBalance(balance))
16       return false;
17     return balance.scale() == 0;
18   }
19   }
20 }
```

Fig. 9: Redefinition of the VR of the property balance

inspector checking the VR for the property balance. An extra constraint is added on top of the constraints defined in the class of bank accounts. The application, now, has two versions of the inspector checking the VR. The pattern must always use the right version. More in particular, the inspector must be invoked against the right 'class object'. *Dynamic binding* ensures using the right version of an instance method. Therefore, an instance method is introduced to retrieve the right 'class object'. Figure 10 illustrates how the right VR inspector is invoked through 'dynamic binding'. Adding new properties to the subclass is

```
1  public class BankAccount {
2    public COBankAccount getClassObject(){
3      return COBankAccount.getInstance();
4    }
5
6    public boolean canHaveAsBalance(
7                        BigDecimal balance){
8      return  getClassObject().
9                isProperValueForBalance(balance) &&
10           getClassObject().
11             isProperBalanceCreditLimit(balance,
12                     getCreditLimit());
13   }
14 }
15
16 public class JuniorBankAccount extends ... {
17   @Override
18   public COJuniorBankAccount getClassObject(){
19     return COJuniorBankAccount.getInstance();
20   }
21 }
```

Fig. 10: 'Dynamic binding' of a 'class method'

now straightforward. If a SR involves a property α from the superclass, the inspector `canHaveAs`α`(T` α`)` needs to be redefined at the level of the subclass. Figure 11 illustrates how the new SR between the properties balance and upper limit is added to the inspector checking the invariant constraints for balance. Lines 5-6 and 13-14 can be generated (O1). Figures 9

and 11 prove that redefinitions are easily developed (O2 - O5). VRs, SRs and TRs can independent of each other be redefined.

```
1  public class JuniorBankAccount extends ...{
2  /**
3   *   @return if (!super.canHaveAsBalance(balance))
4   *           then result == false;
5   *   @return if (!isProperBalanceUpperLimit(balance,
6   *                                 getUpperLimit()))
7   *           then result == false
8   */
9  @Raw @Override
10 public boolean canHaveAsBalance(BigDecimal balance){
11   if (!super.canHaveAsBalance(balance))
12     return false;
13   return getClassObject().isProperBalanceUpperLimit(
14                          balance, getUpperLimit());
15 }
16 }
```

Fig. 11: A SR involving the balance and the upper limit

Pattern skeleton. To summarize, Figure 12 shows a skeleton from the pattern for a property without specification. Given this generated code (O1, O6) the developer has only to (1) complete the inspector checking the VR (2) add an inspector for each SR in the inner class and extend the `canHaveAs`α to invoke the introduced inspector (3) complete the inspector checking the TR.

Language Construct. Figure 12 proves that an inherent problem with patterns is that it generates quite some boilerplate code. The need for patterns signals a lack of expressiveness of programming languages. Therefore, we present an extension to increase that expression power. Figures 13 and 14 illustrate how the example is completely worked out with a new language construct `Property`.

```
1  /**
2   * The balance of this bank account
3   * @Value balance != null
4   * @State balance.compareTo(creditLimit) >= 0
5   * @Trans balance.subtract(new.balance).
6   *           abs().compareTo(MAX_DELTA) <= 0
7   */
8  Property BigDecimal balance isRelatedWith
9                                      creditLimit;
10
11 /**
12  * The credit limit of this bank account
13  * @Value creditLimit != null
14  * @Value creditLimit.signum() < 0
15  */
16 Property BigDecimal creditLimit isRelatedWith
17                                        balance;
```

Fig. 13: The class of bank accounts

The importance of specification is upgraded, by making it an integral part of the construct. The specification describes the different kinds of requirements. They act as guards to validate values in an update operation. Three new tags are introduced to specify the semantics of a `property`, one for each kind of requirement we identified in section IV. The assertions used in the specification are Boolean expressions. (1) Each VR is preceded with a `@Value`-tag. A VR may be split over

```
1  public class Foo {
2
3    public Foo(T α) throws IllegalArgumentException{
4      registerα(α);
5    }
6
7    private T α;
8
9    @Basic @Raw
10   public T getα(){
11     return α;
12   }
13
14   @Raw
15   protected void registerα(T α)
16                 throws IllegalArgumentException{
17     if (!getClassObject().isProperValueForα(α))
18       throw new IllegalArgumentException();
19     this.α = α;
20   }
21
22   @Raw
23   public boolean canHaveAsα(T α){
24     if (!getClassObject().isProperValueForα(α))
25       return false;
26   }
27
28   public boolean canHaveAsNewα(T α){
29     if (!canHaveAsα(α))
30       return false;
31   }
32
33   @Raw
34   public final boolean hasProperα(){
35     return canHaveAsα(getα());
36   }
37
38   public void setα(T α)
39                 throws IllegalArgumentException{
40     if (!canHaveAsNewα(α))
41       throw new IllegalArgumentException();
42     registerα(α);
43   }
44
45   public  COFoo getClassObject(){
46     return COFoo();
47   }
48
49   public static class COFoo
50                       implements ClassObject{
51     private static COFoo instance;
52
53     protected COFoo(){}
54
55     public static COFoo getInstance(){
56       if (instance == null)
57         instance = new COFoo();
58       return instance;
59     }
60
61     public boolean isProperValueForα(T α){
62       return ...;
63     }
64   }
65 }
```

Fig. 12: The pattern for a property α

multiple tags. (2) Each SR is preceded by a `@State`-tag. Each property can be involved in an unlimited number of SRs. (3) Finally, a TR is preceded by a `@Trans`-tag. A SR is always symmetric, which means it applies equal to all properties involved. Despite of this symmetry, the specification doesn't need to be duplicated. Relations between properties need to be mentioned explicitly. The characteristics a property is related with are added to a list following the keyword `isRelatedWith` in the signature of the property. This implies that the specification of the semantics of a property can be spread over multiple properties. We don't consider this as a drawback though because to understand a requirement involving two properties, one has to understand the semantics of both properties anyway. This list also identifies clearly on which properties changes to the specification can have an impact on. By avoiding the duplication we fully support Parnas' principle [8] saying that each fact must be worked out in one, and only one, place. The specification is by definition non-deterministic. The semantics of an assertion Γ in a VR, SR or TR is:

```
if !(Γ)
then result == false
else result == Undefined
```

Thus, when the assertion Γ evaluates to false, the submitted value not acceptable. On the other hand, when the assertion

```
1  /**
2   * The balance of this junior bank account
3   * @Value balance.scale() == 0
4   * @Trans !isBlocked
5   */
6  @Override
7  Property BigDecimal balance isRelatedWith
8              creditLimit,upperLimit,isBlocked;
9
10 /**
11  * The credit limit of this junior bank account
12  * @Value creditLimit.
13  *       compareTo(new BigDecimal(-1000)) >= 0
14  * @Value creditLimit.scale() == 0
15  */
16 @Override
17 Property BigDecimal creditLimit isRelatedWith
18                                     balance;
19
20 /**
21  * The blocked state of this ...
22  */
23 Property boolean isBlocked isRelatedWith
24                                     balance;
25
26 /**
27  * The upper limit of this bank account
28  * @Value upperLimit >= 1000
29  * @Value upperLimit <= 10000
30  * @State balance.compareTo(
31  *           new BigDecimal(upperLimit))<=0
32  */
33 @Immutable
34 Property int upperLimit isRelatedWith
35                                     balance;
```

Fig. 14: The class of junior bank accounts

evaluates to true the value may be acceptable. The semantics of the VRs of credit limit in Figure 13 is that non-effective positive or zero decimal numbers are certainly not a good value for a credit limit. Negative values *can* be good values. Subclasses are allowed to further specify the *open* part. The requirements specified in a subclass are *added* to the requirements specified in the superclass. The VR of the credit limit in the class of junior bank accounts for instance now specifies that only strictly negative integer numbers are acceptable values.

Evaluation. Up to now, the pattern has only been applied to academic problems. These experiments show that about 70% of the code for defining properties is boilerplate code. As an example the full definition of class of bank accounts counts 360 lines of Java code. About 250 of these lines are boilerplate code. The typical Java programmer is not tempted to write all these lines in original definitions of classes. In particular, he will not be eager to encapsulate the different kind of requirements in Boolean inspectors such as `isProperValueForBalance()`, `canHaveAsBalance()`, `canHaveAsNewBalance()`, etc. This either leads to duplicate code because the same requirement is repeated over and over again in different parts of the class definition, or it compromises adaptability in time and space. We therefore believe that more advanced language constructs are needed to introduce properties in classes. We still need to experiment with this pattern in the scope of industrial software systems. We expect the same results with respect to the mere definition of properties in such large systems. The pattern gives the programmer the opportunity to focus more on the business at stake.

VI. Related Work

The central idea of Model Driven Architecture (MDA) [11], [12] is to automate transformations between models. To enable this transformations the specification should be defined in a formal way. MDA uses Design by Contract (DBC) [13] to specify the semantics of models formally. DBC was developed by Bertrand Meyer as part of the Eiffel programming language [1], [18]. DBC is based amongst others on Hoare-logic [7] that already introduced concepts like *preconditions* and *postconditions*. Other object-oriented languages with native support for DBC are for instance Sather [20], Nice [19] and Spec# [21]. Commonly used languages like Java, C++ [14] and C# [22] have no support for DBC. However, several third-party tools have been developed for those languages. Tools for java are for example: Contract4J [23], JContractor [24]. The Java Modeling Language [25] is a behavioral interface specification language that can be used to specify the behavior of Java modules. B AMN [26] and UML-RSDS [27] present similar concepts. In UML-RSDS correct operations can be synthesized from invariants (VR and SR constraints in this paper) in many cases. In B, a TR can be expressed as an abstract pre-post specification which is correctly refined by a more concrete operation that ensures the TR constraints.

VII. Conclusion and Future Work

In Object-Oriented programming, a significant effort has been made in recent years to increase the expressiveness of programming constructs for the production of code. However, we have not seen such a similar evolution in the design and

specification of code. Developers are often forced to write ad-hoc code specifications in a non-standardized manner. In this paper, we therefore took an evolutionary approach to language-integrated specification constructs. We started from existing specification constructs (@pre, @post, ...) with the ambition to enhance the overall expressiveness of specifications in object-oriented languages.

We have identified three types of requirements that can occur in program specifications: Value Requirements (VR), State Requirements (SR), and Transition Requirements (TR). **Value Requirements** are used to specify the most basic kind of business rules in that they restrict the range of values that a characteristic, property or association, can have. A **state requirement** describes a constraint that restricts acceptable value combinations of (a set of) properties. **Transition Requirements**, then, specify the business rules that restrict the evolution of property values.

Besides that, we feel that also need to help developers in correctly using these constructs. Therefore, we have introduced a "boilerplate pattern" that showcases the inspector methods required for validating the VR, SR and TR in a specification. But a pattern is, according to us, not sufficient as a solution, because (1) it involves too much boilerplate code and (2) there remains a risk of incorrectly implementing (a part of) the pattern, which would still lead to ill-defined specifications.

Therefore, we have integrated a Specification Language extension in Java. By means of the @value, @state and @trans tags, developers can better capture the Specification of their code, while outsourcing all technicalities to a code generator. We have also introduced the **isRelatedWith** construct in order to further minimize the risk of duplicate specifications. As an additional benefit, the formal specifications are compile-time checked, since they are injected in the Java code in the background, before compilation.

We recognize that this is the first step in our roadmap to develop a fully integrated, expressive Specification language, and would like to conclude by giving the reader a view of our upcoming research, which has two important future directions. Next to our Specification-to-Code generator, we also want to build a detailed formalization of the Specification language, in order to identify opportunities to further enhance the expressiveness of the concepts. A second direction is to further increase the expressiveness and action radius of the concepts. For instance, we are currently working to add determinism to our Specification constructs, for which a prototype definition is currently available, but too preliminary for this paper. Another example is that we are defining more finegrained rules on when properties may be added or removed to the isRelatedWith-list. These rules are currently still based on informal guidelines.

Acknowledgements We especially thank Sven De Labey for his advice. We thank the anonymous reviewers for their insightful comments. This research is funded by the the Fund for Scientic Research (FWO) in Flanders.

REFERENCES

[1] B. Meyer, *Object-oriented software construction*, second edition ed., Prentice Hall, 1997.

[2] E. Gamma, and R. Helm, and R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] B. H. Liskov and J. M. Wing, *A behavioral notion of subtyping*, ACM Trans. Program. Lang. Syst. 0164-0925 (1994), pp. 1811–1841.

[4] J. Bloch, *Effective java*, Java Series, Pearson Education, 2008.

[5] J. Bosch, *Design patterns as language constructs*, Journal of Object-Oriented Programming 11 (1998), pp. 18–32.

[6] M. Feathers, *Working effectively with legacy code*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[7] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM 12 (1969), no. 10, pp. 576–580.

[8] D. L. Parnas, *On the criteria to be used in decomposing systems into modules*, Commun. ACM 15 (1972), no. 12, pp. 1053–1058.

[9] E. Steegmans, *Object oriented programming with java*, Acco, 2011.

[10] A.L. Rubinger and B. Burke, *Enterprise javabeans 3.1*, O'Reilly Media, 2010.

[11] D. Frankel, *Model driven architecture: Applying mda to enterprise computing*, OMG Press, Wiley, 2003.

[12] A. G. Kleppe, and J. Warmer, and W. Bast, *Mda explained: The model driven architecture: Practice and promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] R. Mitchell, and J. McKim, and B. Meyer, *Design by contract, by example*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[14] S. B. Lippman, and J. Lajoie, and B. E. Moo, *C++ primer*, 5th edition ed., Addison-Wesley Professional, 2012.

[15] Javadoc Tool Home Page, *http://java.sun.com*, retrieved: 08, 2013

[16] Enterprise JavaBeans Technology, *http://java.sun.com*, retrieved: 08, 2013

[17] Eclipse project, *http://www.eclipse.org*, retrieved: 08, 2013

[18] Eiffel Software, *http://www.eiffel.com/*, retrieved: 08, 2013

[19] The Nice Programming Language, *http://nice.sourceforge.net*, retrieved: 08, 2013

[20] Sather, *http://www1.icsi.berkeley.edu/ sather*, retrieved: 08, 2013

[21] Microsoft Research Spec#, *http://research.microsoft.com/en-us/projects/specsharp*, retrieved: 08, 2013

[22] C# Programming Guide, *http://msdn.microsoft.com/en-us/library/vstudio/67ef8sbd.aspx*, retrieved: 08, 2013

[23] Contract4J, *http://www.polyglotprogramming.com/contract4j*, retrieved: 08, 2013

[24] M. Karaorman, and U. Holzle, and J. Bruno, *jcontractor: A reflective java library to support design by contract*, Tech. report, Santa Barbara, CA, USA, 1999.

[25] G. T. Leavens and Y. Cheon, *Design by Contract with JML*, http://www.eecs.ucf.edu/ leavens/JML//jmldbc.pdf, retrieved: 08, 2013

[26] J.-R. Abrial, *The B-book - assigning programs to meanings*, Cambridge University Press, 2005.

[27] K. Lano and S. K. Rahimi, *Synthesis of Software from Logical Constraints*, ICSOFT, 2012, pp. 355-358.